ECE438
Design of Computers

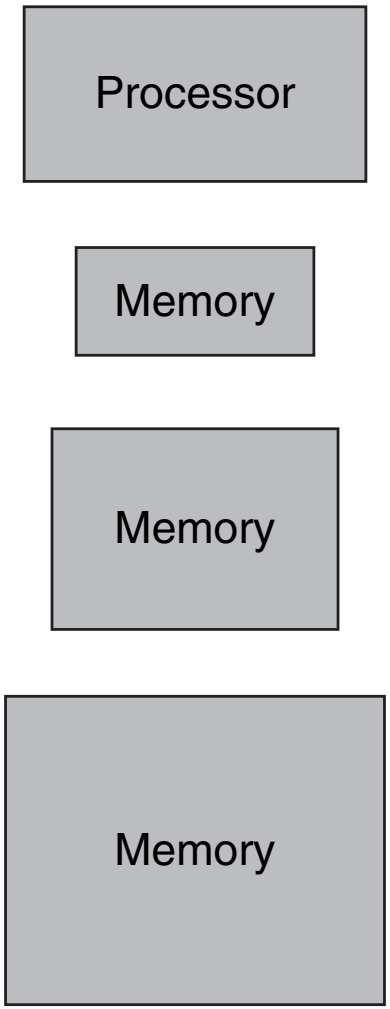
Instructor: Andrew David Targhetta

April 20, 2020

Introduction

- The primary focus of chapter 5 is memory hierarchy
- Programs demand more memory and higher performance
- Large memories are slow to access
- Solution?

The Memory Hierarchy



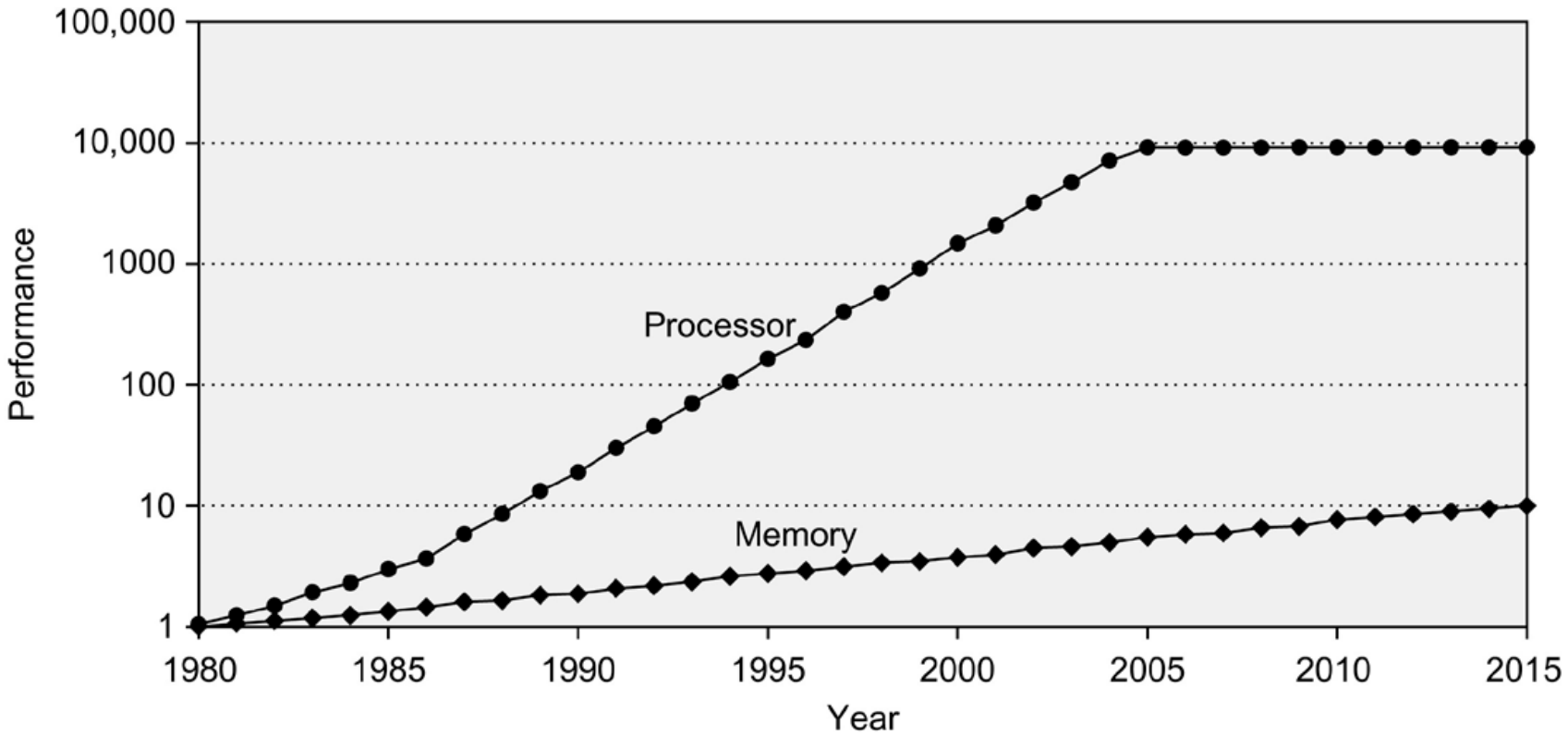
The Memory Hierarchy

- Use small, fast SRAMs to *cache* data and instructions
- Memory closest to the processor is the smallest and fastest
- Memory becomes larger and slower as it approaches main memory
- Main memory uses DRAM for density

Memory/processor performance gap

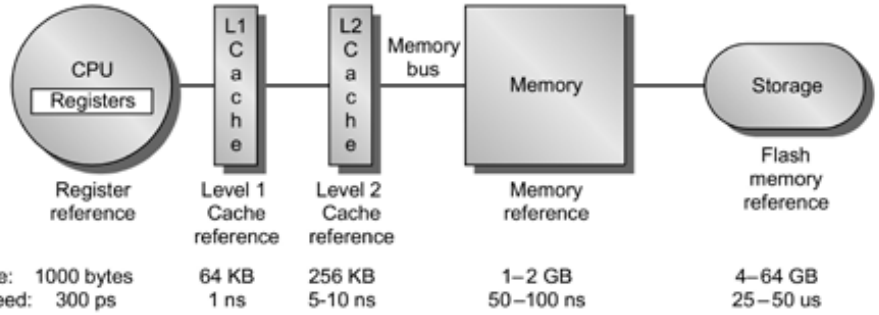
- In the 70s, processors could execute directly out of DRAM
- The Apple II for example, had 64kB of main memory and no cache
- In the early 80s, processor performance started out pacing memory performance

Memory/processor performance gap

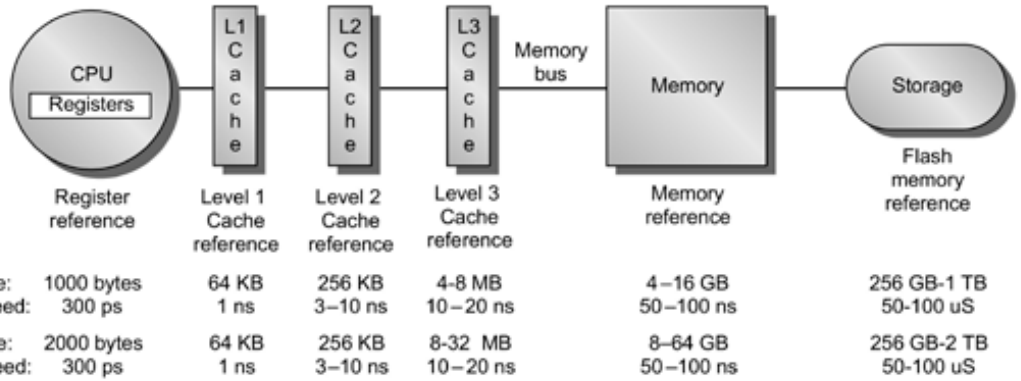


Notes about the prior graph

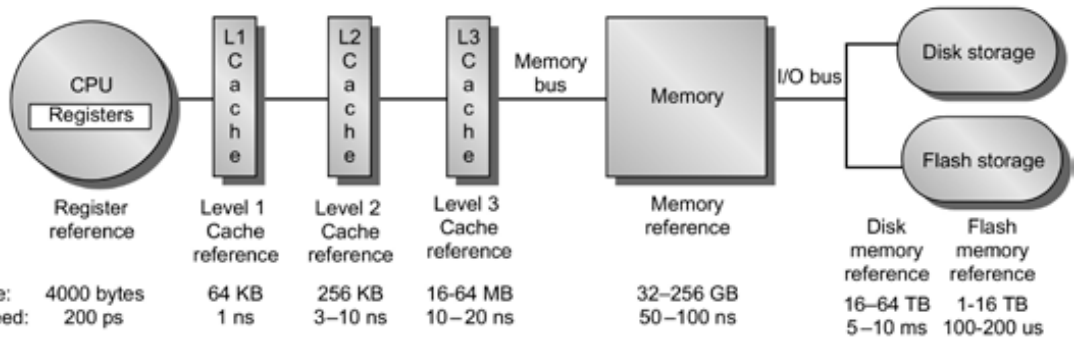
- Memory performance is in terms of latency
- The processor performance leveled off because it was for single-core only!
- What would multi-core do to the amount of required memory bandwidth?



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

How does this even work?!?

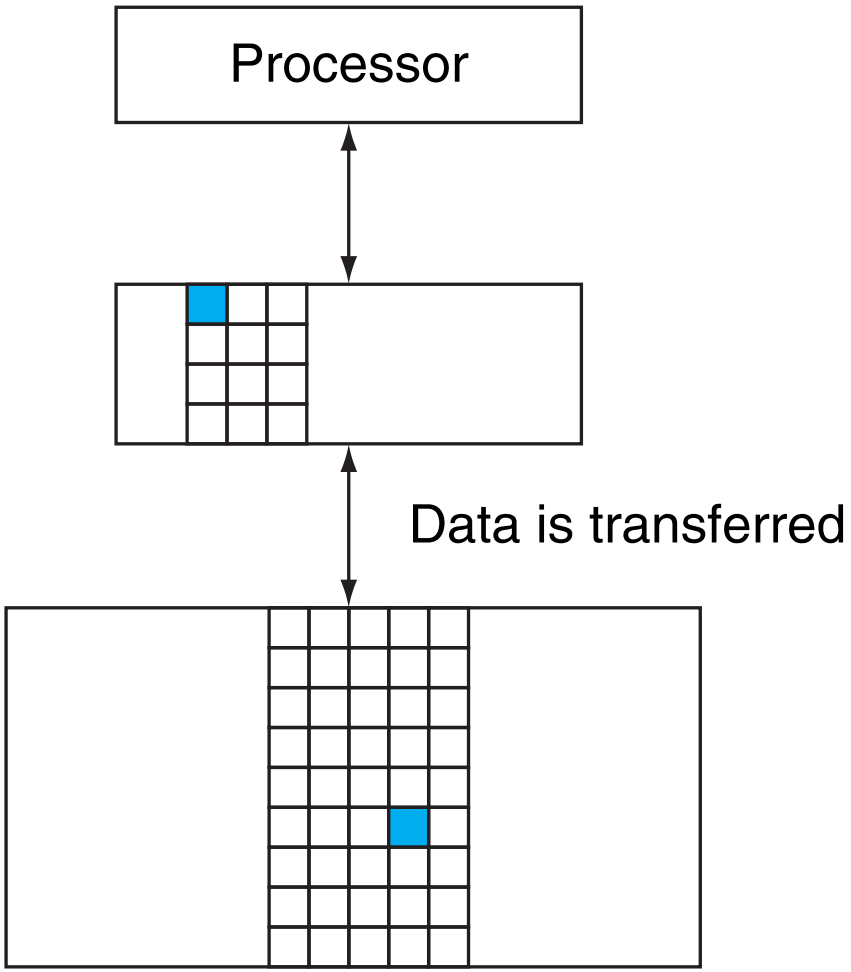
- Caching is based on the *principal of locality*
  - *Temporal locality* states that if a data location is referenced, there is a high likelihood it will soon be referenced again
  - *Spacial locality* state that data location is referenced, there is a high likelihood that nearby locations will be referenced as well
- Programs exhibit significant amounts of temporal and spacial locality

locality in programs

- The way we fetch instructions sequentially leads to *spacial* locality
- Loops in our programs lead to *temporal* locality
- Array access creates *spacial* locality

How does caching exploit locality?

- The cache (i.e. smaller SRAM) keeps recently used data and instruction to exploit *temporal* locality
- Access to this *cache* is fast because it's small
- Data is transferred between the cache and the next level in the hierarchy (main memory assuming 1 level of cache) in *blocks*, exploiting *spacial* locality and taking advantage of the *bandwidth* of next level


Processor
Data is transferred

Caching explained

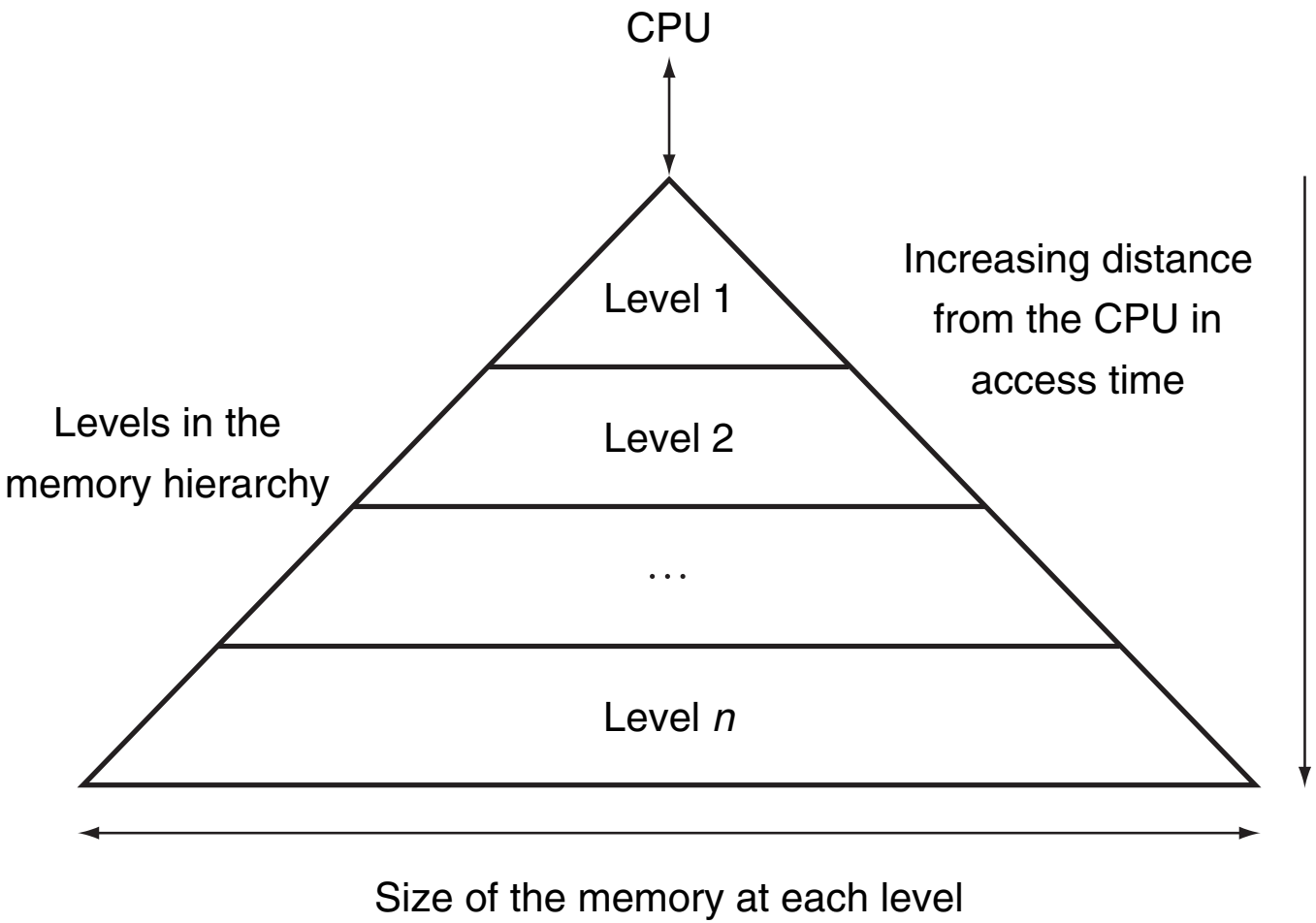
- The processor directly accesses the level-1 cache
- If the data the processor is requesting is in the level-1, a *cache hit* happens and the cache delivers the data to the processor
- If the data is not in the level-1, a *cache miss* happens
  - The processor waits for the data
  - The cache requests data from memory, delivers data to processor when it arrives, and keeps the block for later use

Some cache terms

- *hit rate* is the ratio of cache hits to the number of accesses to the cache
- *miss rate* the ratio of cache misses to the number of accesses to the cache

Extend to multiple levels

- Once you understand a single-level memory hierarchy, it's very easy to extrapolate the concept out to multiple levels
- The level 2 (L2 for short) will be larger but slower than the L1
- The L3 will be larger and slower than the L2 and so on and so forth


CPU
Level 1
Level 2
...
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Memory technology

- SRAM is used for L1 through L3 because it is fast
- DRAM is used for L4 and main memory because it is dense and cheap
- Flash and magnetic disk are used for storage because they are non-volatile and extremely dense

Access times

- SRAM — 0.5-2.5ns
- DRAM — 50-70ns
- Flash — 5,000-50,000 ns
- Magnetic disk — 5,000,000-20,000,000 ns

The history of DRAM

Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

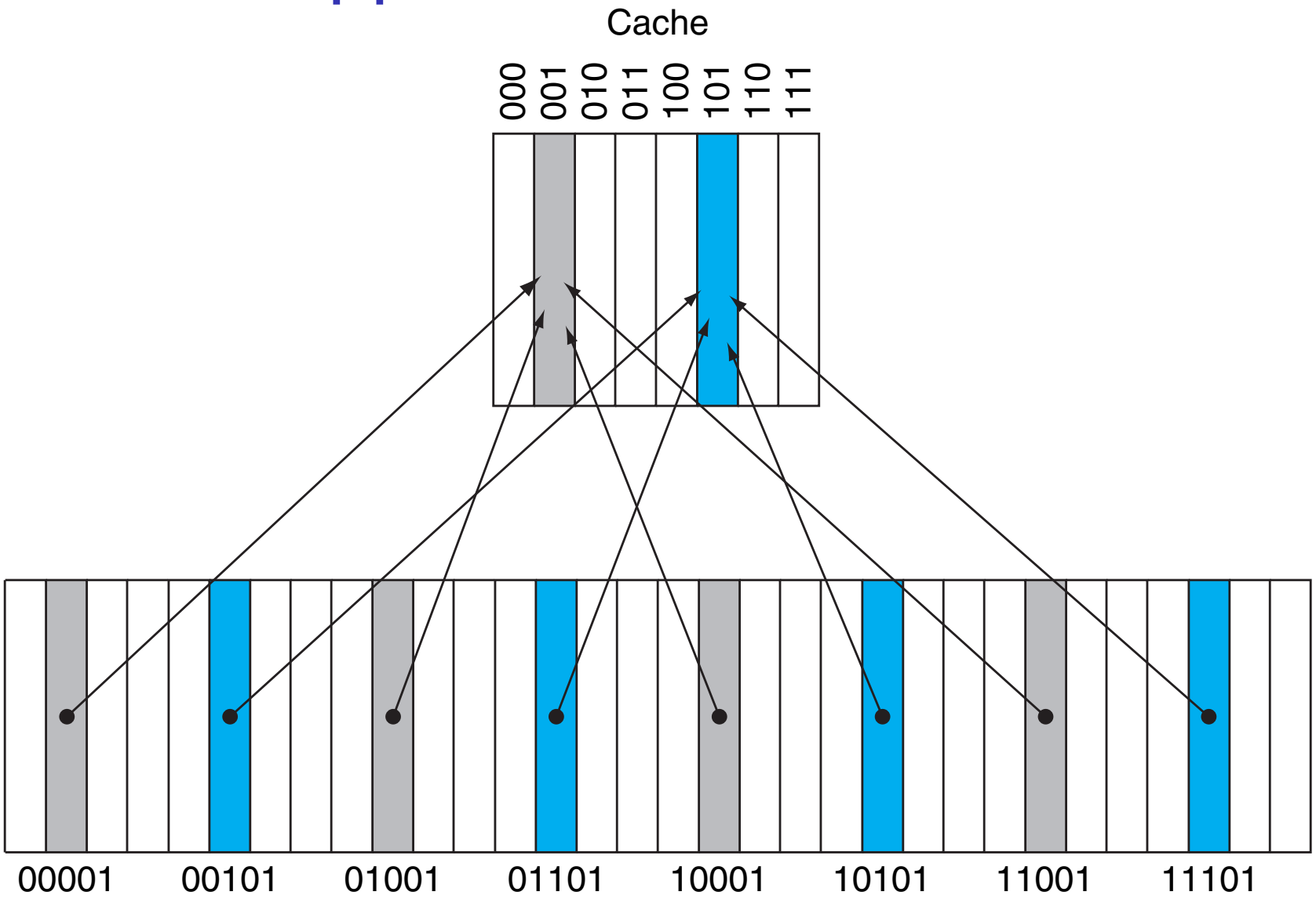
The Basics of Caches

- Generally *inclusive*
  - This means if the data is in the L1 cache, it is also in the L2. If it's in the L2, it's in the L3, and so on and so forth
  - Implies that the $L1 << L2 << L3 << memory$ in terms of size
- For now, let's just assume one level of cache
- The simplest design is a *direct-mapped* cache

The direct-mapped cache

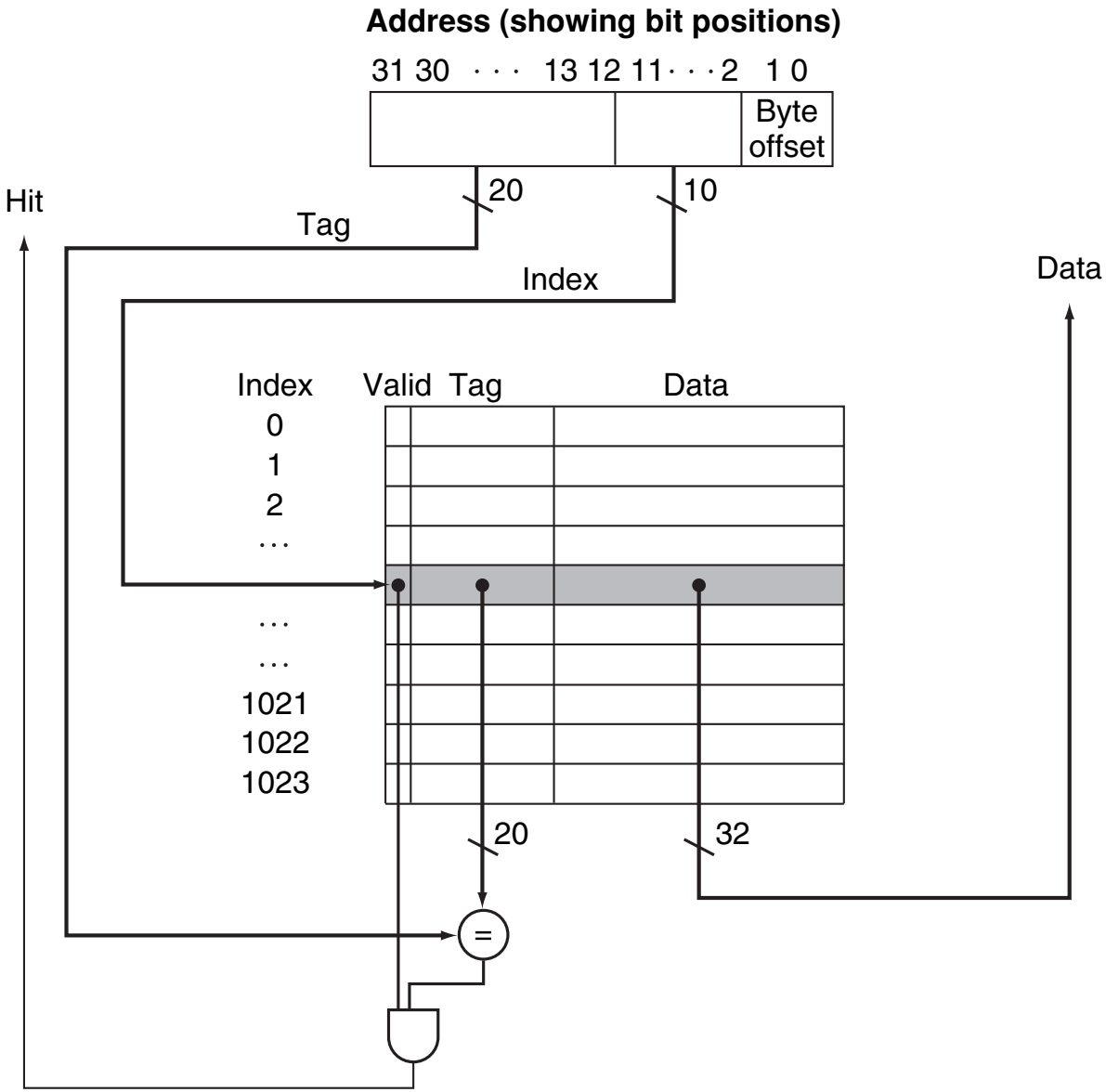
- Each memory location maps to only one entry in the cache
- The lower address bits serve as the *index* into the cache
- The upper address bits are used as the *tag* to uniquely identify a *block*

A direct-mapped cache



A simple hardware example

- For now, let's assume a cache block is only 32 bits, i.e. one word, and there are 1024 blocks in our cache
- The lower 2 bits of the address make up the byte offset
- The next 10 bits of the address make up the index
- The remaining bits (20) make up the tag
- A valid bit is stored with the tag

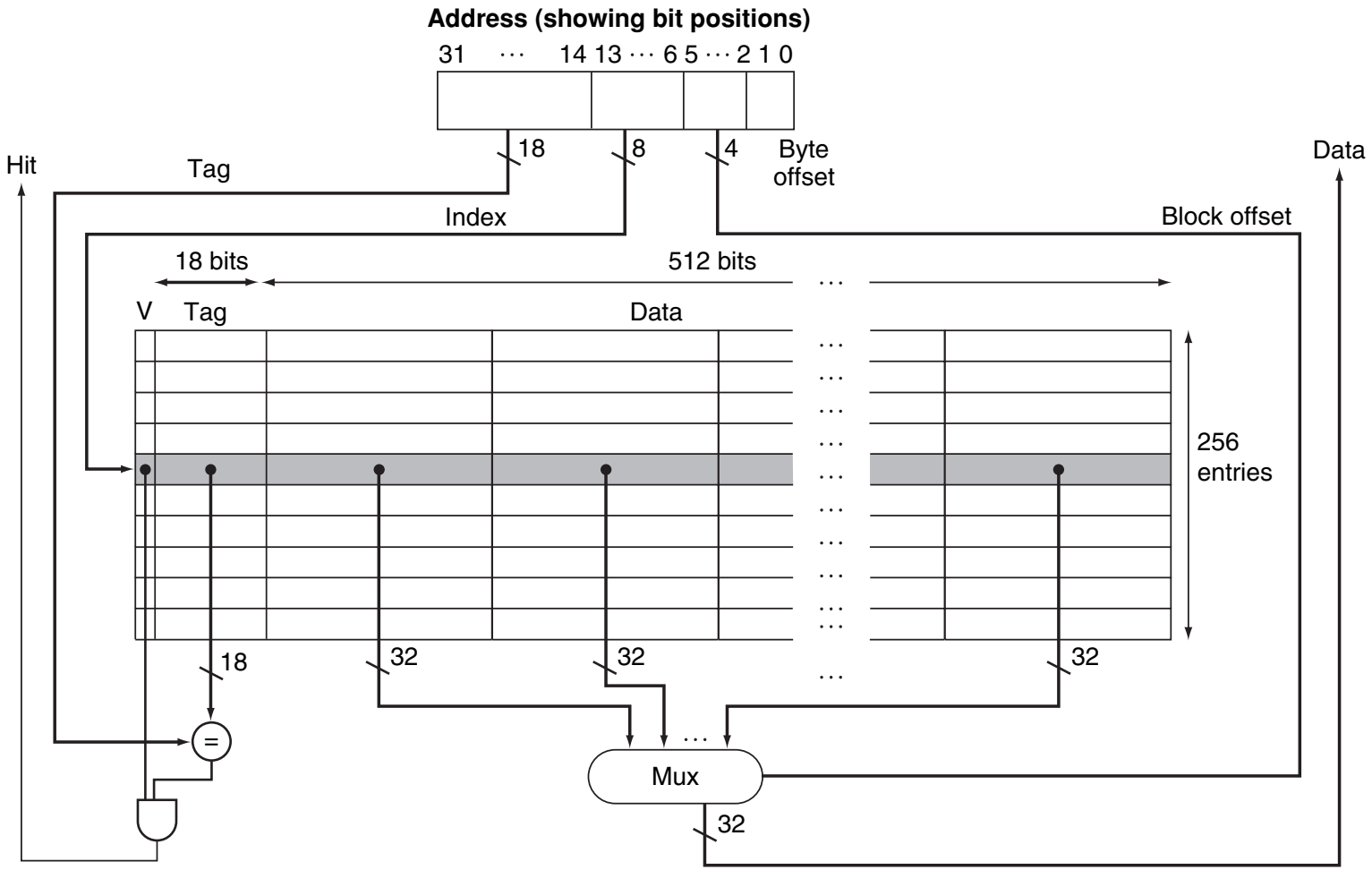

Address (showing bit positions)
31 30 · · · 13 12 11 · · · 2 1 0
Byte offset
20
10
Hit
Tag
Index
Data
Index
Valid
Tag
Data
0
1
2
· · ·
· · ·
· · ·
1021
1022
1023
20
32
=

Walk through an example...

- Figure 5.9 of book provides an example of accessing the cache
- Assumes an 8-block cache with 1 word per block
- uses a 5-bit *word* address

Block size explained

- Each cache *block* (a.k.a. a cache *line*) actually contains multiple *words*
  - Remember *spacial locality*??
  - A machine with a 32-bit word size and a 64-byte cache block has 16 words per cache block
  - A 4-bit block offset is used to select a word from a block
- Extending our simple example to include larger block sizes is easy


Address (showing bit positions)
31 ··· 14 13 ··· 6 5 ··· 2 1 0
18
8
4
Byte offset
Hit
Tag
Index
Block offset
Data
18 bits
512 bits
V
Tag
Data
256 entries
18
32
32
32
=
Mux
32

The previous hardware example...

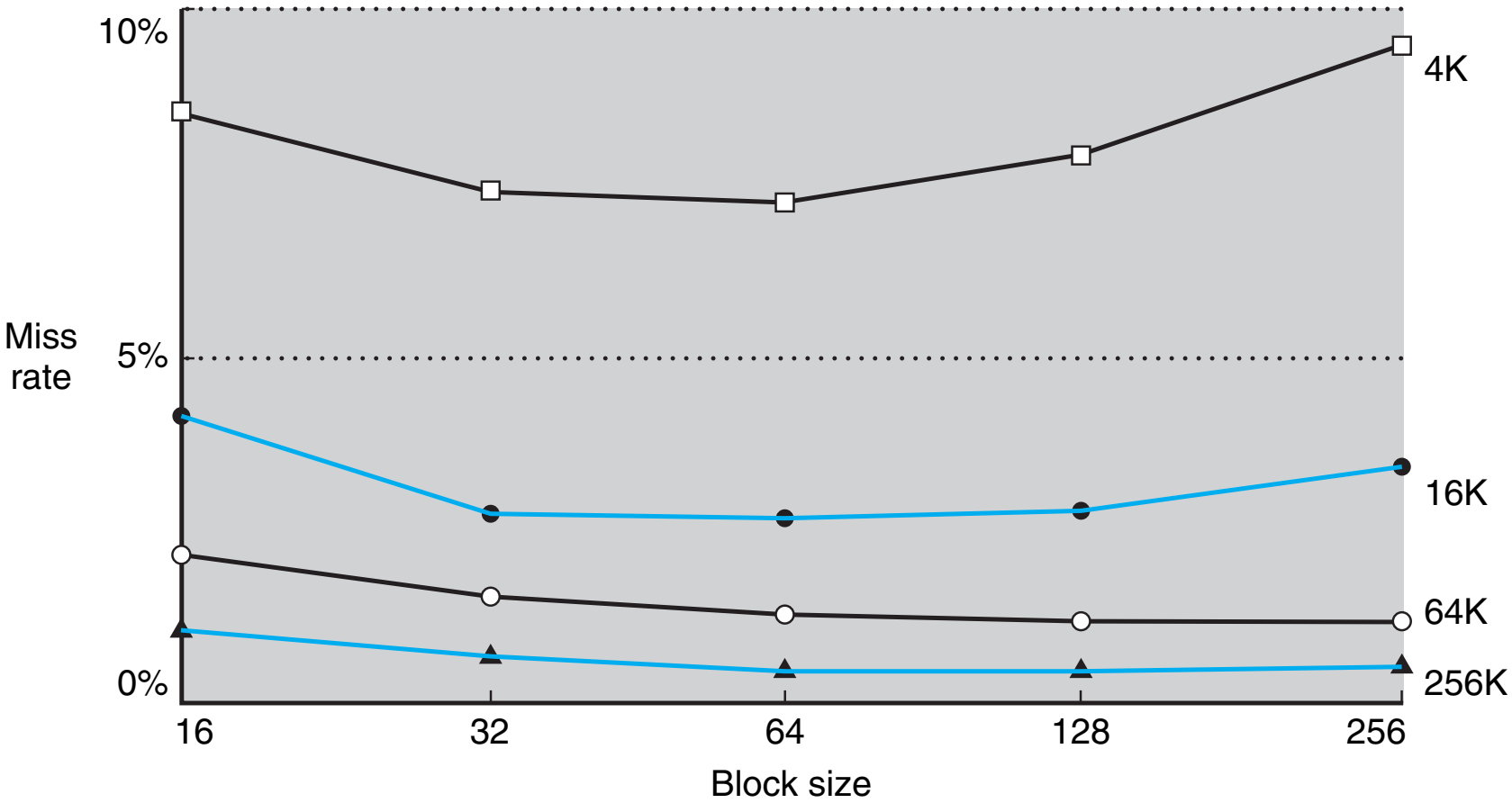
- Was in fact from a real processor, the Intrinsity FastMATH
- It has 256, 64-byte cache blocks, for a capacity of 16kB
- Separate 16kB caches were used for instructions and data, which is referred to as *split* caches as opposed to *unified*

How well does it work??

- This cache achieved a 0.4% miss rate for instructions and 11.4% for data on SPEC2000 benchmarks
- So why does the instruction cache perform better??
- The effective combined miss rate is 3.2%

Let's look at a graph!

- Using SPEC92 as a benchmark and a DECstation 5000
- Vary block size and cache size
- Plot effective miss rate


10%
5%
0%
Miss rate
4K
16K
64K
256K
16
32
64
128
256
Block size

Classification of cache misses

- To better understand cache performance and how to improve it
- We can divide cache misses into the 3 C's
  - *Compulsory*: initial cache misses due to a *cold* cache (i.e. empty)
  - *Capacity*: cache misses due to the cache size being too small
  - *Conflict*: cache missed due to the cache not being *fully associative*

Reducing cache misses

- Increase the block size to reduce compulsory misses
- Increase the cache size to reduce capacity misses
- Increase the associativity to reduce conflict misses

No free lunch!

- Increase the block size
  - Can increase conflict misses!
  - Increases time to handle miss
- Increase the cache size
  - Requires more area
  - Slows the cache down
- Increase the associativity
  - Increases complexity
  - Slows cache down

Okay so what about writes?!?

- 2 different strategies for handling writes
  - *Write-through*: always write data to next level in hierarchy
  - *Write-back*: only write data to next level during a block replacement
- In either case, a *write-buffer* can improve the performance by not stalling the processor while data is being written to the next slower level

A closer look at *write-through*

- The advantages
  - Simple to implement
  - Keeps the next level of memory consistent
- The disadvantages
  - A major performance bottleneck because the processor must stall on multiple writes
  - Creates congestion on bus and next level of memory

A closer look at *write-back*

- The advantages
  - Much better for performance as writes do not go to the next level until the block is being *evicted*
  - Significantly reduces congestion on bus and next level of memory
- The disadvantages
  - The cache controller is more complex
  - Maintaining consistency with next level of memory is more complex
  - Increases read miss penalties. What?!? How?

The *write buffer* helps either way

- In the case of write-through, writes go to the buffer from the processor and avoid processor stalls as long as the write buffer is not full
- In the case of write-back, writes go into the buffer when a block is being evicted allowing the cache controller to immediate begin retrieving the requested cache block

Write allocate vs. *no write allocate*

- In addition to write-back vs. write-through, we must choose between write allocate and no write allocate
- Write allocate: Write misses force the cache to retrieve the block being written to from the next level of memory
- No write allocate: Do not retrieve the block from the next level of memory unless there is a read miss

Why does it matter?

- Some programs will simply write to data and never read it
- In such cases, no write allocate is an optimization that avoids unnecessary reads from the next level of memory
- Most cache controllers allow all of these write strategies to be programmed on the fly

Processor performance and the memory system

- In a simple processor, the pipeline must stall when the cache is handling a *miss*
- A *hit* is assumed to cause no interruption in the flow of instructions
- We can extend our performance analysis:

$$\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) * \text{cycle time}$$

Memory stall cycles

- In terms of miss rate:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} * \text{Miss rate} * \text{Miss penalty}$$

- In terms of misses per instruction:

$$\text{Memory stall cycles} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Misses}}{\text{Instruction}} * \text{Miss penalty}$$

Separate out reads and writes

- The previous equations assumed reads and writes have the same cost
- Read stalls: Read stall cycles $=$

$$\frac{\text{Reads}}{\text{Program}} * \text{Read miss rate} * \textit{Readmisspenalty}$$

- Write stalls:

$$\text{Write stall cycles} = (\frac{\text{Writes}}{\text{Program}} * \text{Write miss rate} * \text{Write miss penalty}) + \text{Write buffer stalls}$$

But what about out-of-order processors?

- For complicated, out-of-order processors, equating memory system performance to processor performance is much more difficult
- OoO processors are able to hide level 1 cache misses by executing other instructions while the cache miss is serviced
- Thus, we want a different way to calculate memory performance

Average Memory Access Time

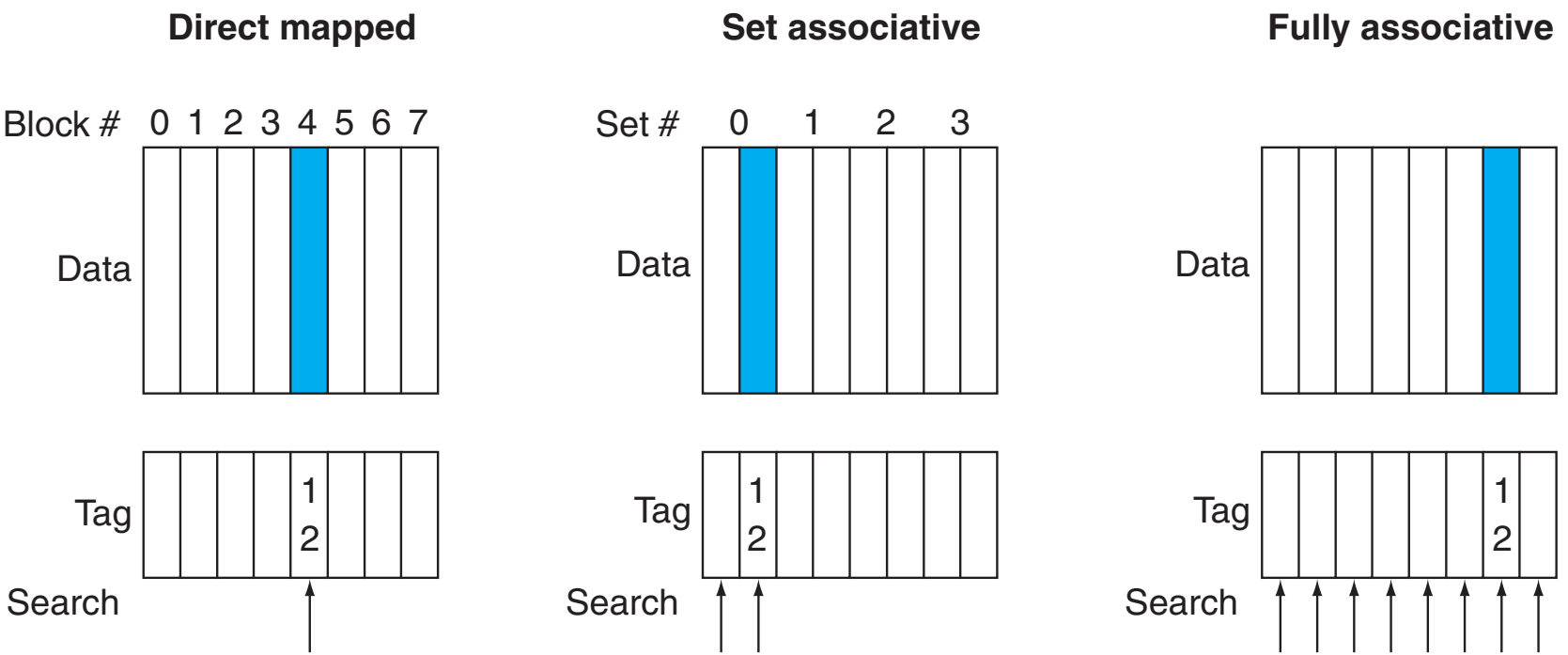
- $\text{AMAT} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty}$

A more flexible placement of blocks...

- We can reduce cache misses by adding associativity
- Recall:
  - *Direct mapped* has only one location in the cache that a block can be placed
  - *Fully associative* allows blocks to be placed anywhere in the cache
- So is there a middle ground?? Yes!

The middle ground

- A *set associative* cache has a fixed number of locations greater than one that a block can be placed
- A 2-way set associative cache provides two *ways* with a *set* that a cache block can be placed
- For example, a 2-way set associative cache with 8 blocks has 4 sets, each with two ways


Direct mapped
Block # 0 1 2 3 4 5 6 7
Data
Tag
1
2
Search
Set associative
Set # 0 1 2 3
Data
Tag
1
2
Search
Fully associative
Data
Tag
1
2
Search

So how is a set-associative cache organized?

- Each block has its own *tag*
- The *sets* are selected with the *index*
- Adding more associativity reduces the size of the index and increases the size of tag because it reduces the number of sets

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

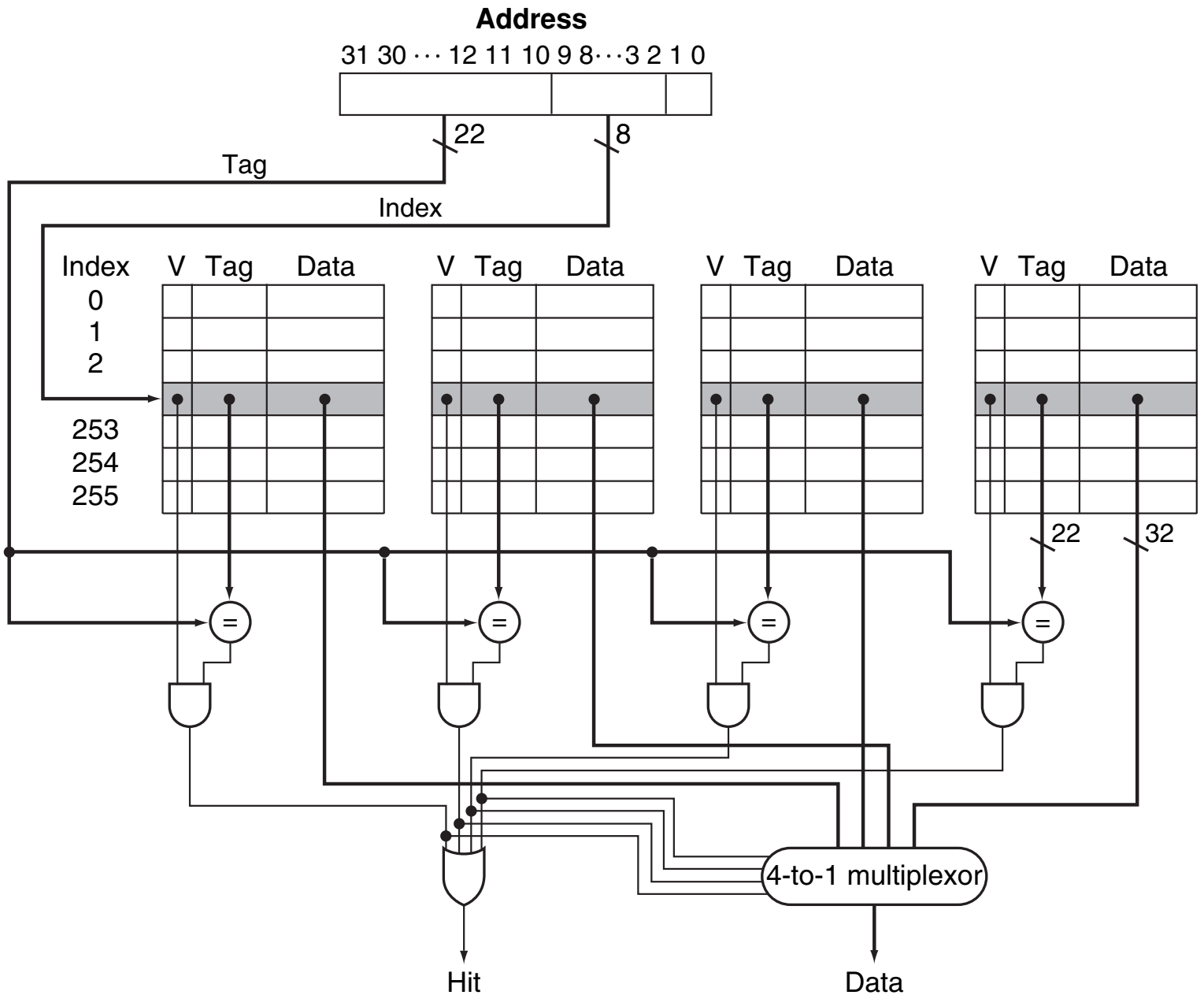
Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

How do we construct set-associative caches??

- Easily actually! Just use multiple direct-mapped caches with a mux to select between the *ways*
- Consider an example 4-way set associative cache with 256 sets


Address
31 30 ··· 12 11 10 9 8···3 2 1 0
22
8
Tag
Index
Index
V
Tag
Data
0
1
2
253
254
255
22
32
=
4-to-1 multiplexor
Hit
Data

The trade offs of associativity

- The primary advantage of set-associative caches is a reduction in conflict misses
- Disadvantages include:
  - Increased *hit time* due to extra comparison logic
  - Pipelining difficulties due to more than one way
  - Increased energy consumption due to reading more data that won't be used

So how well does it work??

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

- SPEC CPU2000, Intrinsity FastMATH
- 64kB data cache with 64-byte blocks
- 15% decrease from direct mapped to 2-way
- Not much advantage past 2-way for this example

Let's look at an example...

- We will work through the example on page 405 of the book on the board

Which block to replace?!?

- Associativity introduces another question: When replacing a block within a set, which block should we *evict*?
- A common scheme is Least Recently Used (LRU)
- LRU is easy with 2-way set associative but gets harder as the associativity increases
- thus, we have pseudo-LRU and random, which actually works well

More on implementation

- 8-way and beyond becomes more difficult to implement with just SRAMs and comparators
- We can combine the SRAM and comparator into one cell to create *Content Addressable Memory* (CAM)
- CAMs can be used for tag storage which will detect a hit and provide the index of the data, which can then be fed into an SRAM for the data store

Now on to *Virtual Memory*!

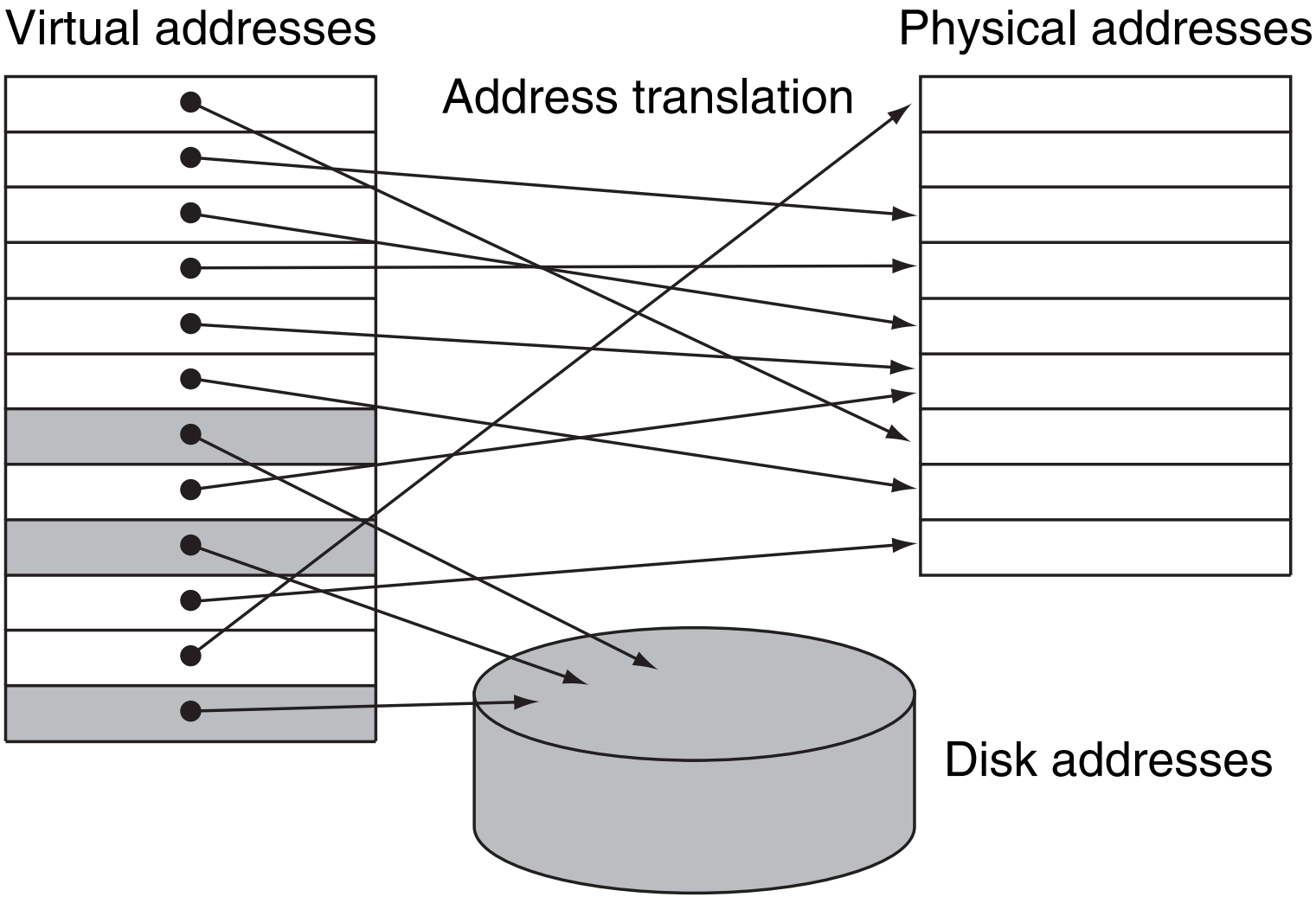
- Was historically developed to cope with small amounts of main memory, swapping data between disk and main memory automatically
- What else does *virtual memory* provide
  - Allows multiple processes to share resources easily
  - Each *process* or program has its own *virtual* view of memory
  - Protection between various processes and between processes and the Operating System (OS)

Some virtual memory terminology

- Don't confuse virtual memory with a *virtual machine*(VM)!
  VMs take virtualization to a whole new level, virtualizing the entire machine as opposed to just the memory
- *Physical* address space is that of the actual machine
- *Virtual* address space is that of the *process* or program

A mapping of address

- Virtual memory provides an automatic mapping from virtual addresses of the program to physical addresses of the machine
- Each virtual address space is broken up into *pages*, which can be located almost anywhere in physical memory or on disk
- Storing pages on disk allows a program's virtual address space to be larger than the machines physical address space!

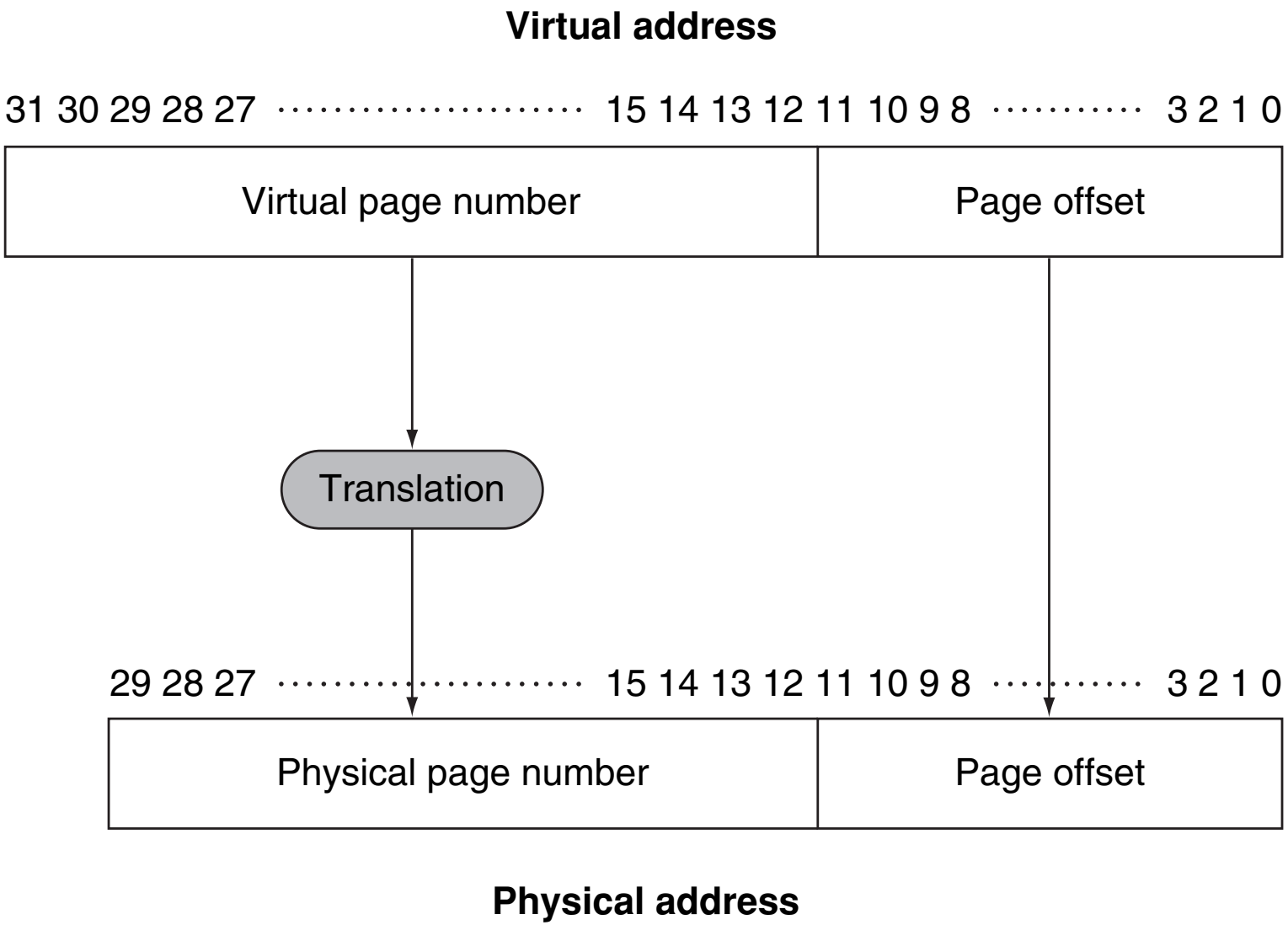

Virtual addresses
Physical addresses
Address translation
Disk addresses

Multiple virtual address spaces

- The address mapping (a.k.a *address translation*) allows for multiple virtual address memories
- Each *process* has its own virtual memory!
- This provides memory *protection* from one virtual address space to another, i.e. process A cannot access process B's memory unless it is specifically shared

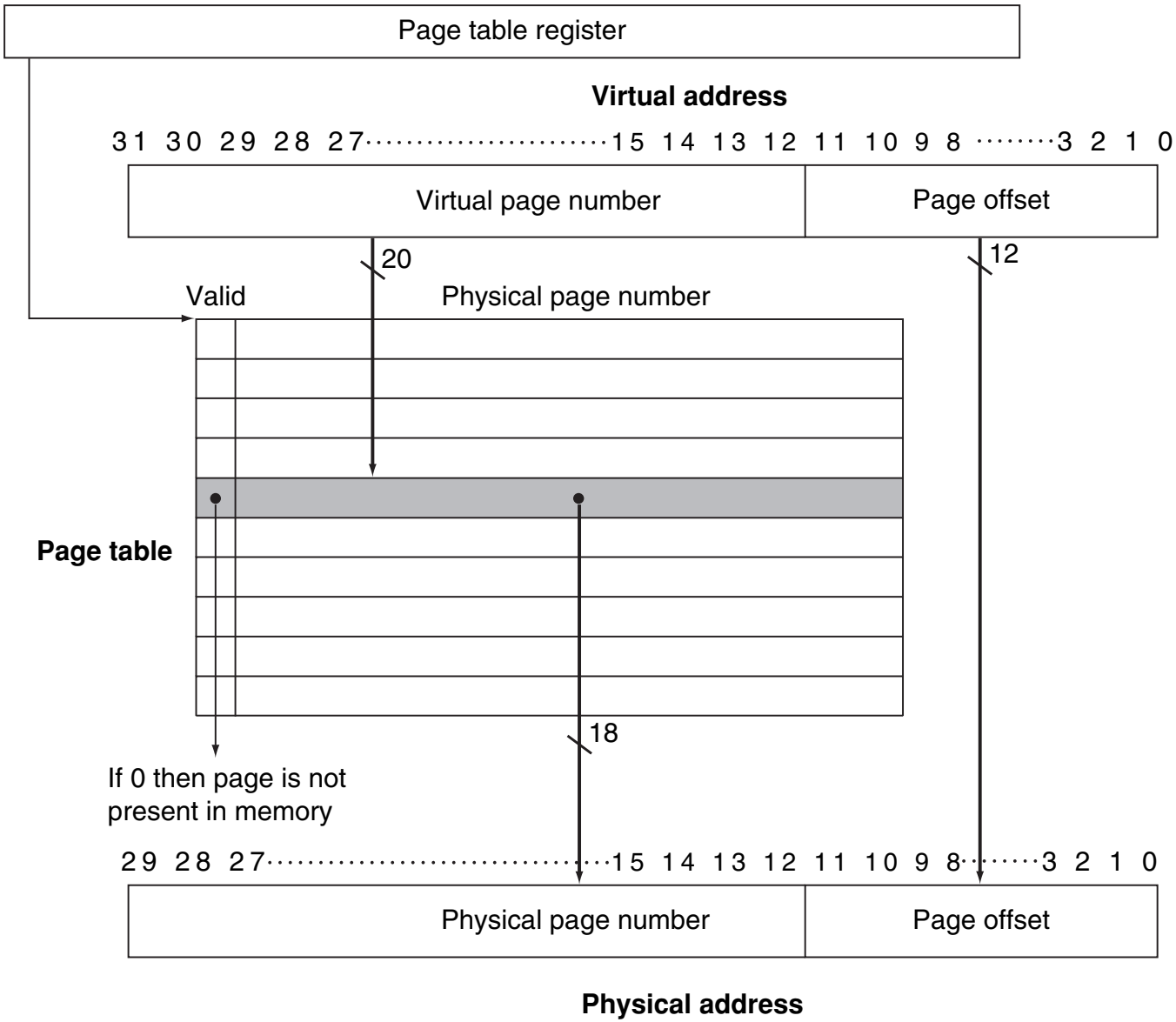
Pages and address translation

- To efficiently translate addresses, the virtual address space is divided into pages (typically 4kB in size but usually adjustable depending on the OS)
- Thus, the virtual address is broken up into *virtual page number* and *page offset*
- The virtual page number is then translated into a *physical page number*


Virtual address
31 30 29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Virtual page number
Page offset
Translation
29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Physical page number
Page offset
Physical address

So how does this translation happen??

- A page table that is maintained by the OS keeps track of this address mapping
- Each process has its own page table pointed to by a *page table register*
- The page table is then indexed by the virtual page number

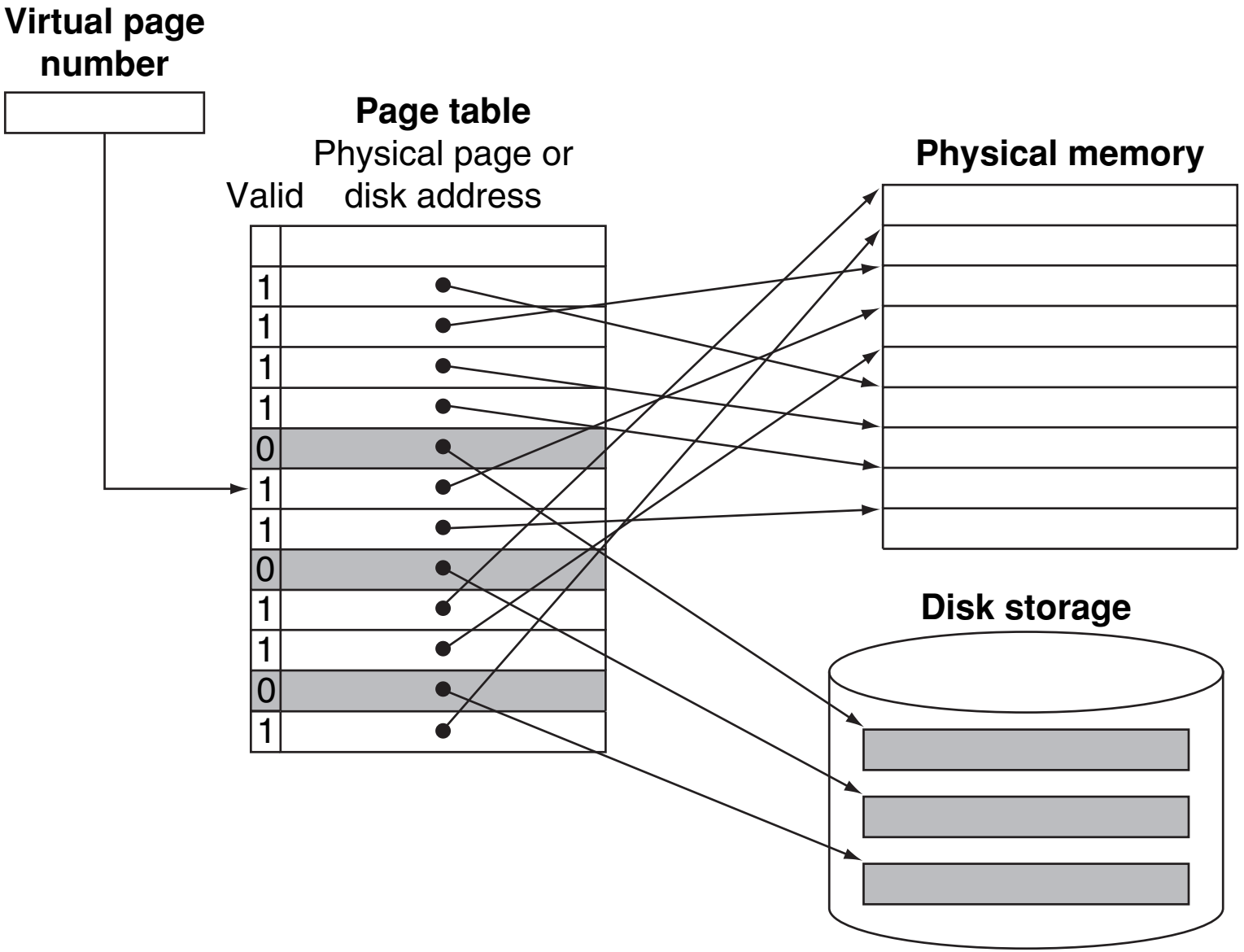

Page table register
Virtual address
31 30 29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
18
If 0 then page is not present in memory
29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Physical page number
Page offset
Physical address

The page table

- The size of the page table can get quite large!
- If we have a 32-bit virtual address space and 4kB pages, we would need 2^{20} or 1 million page table entries!
- Each entry would be 32 bits so that is 4MB for each page table
- Virtual memory systems will typically use a multi-level page table to reduce the size of the page table

Swap space and the valid bit

- Each page table entry has a valid bit that indicates the page is in physical memory
- If the valid bit is not asserted, the page is either on disk or doesn't exist
- A *page fault* refers to the situation when a requested page is not in physical memory and the OS must retrieve it from the location on disk called *swap space* where pages are stored

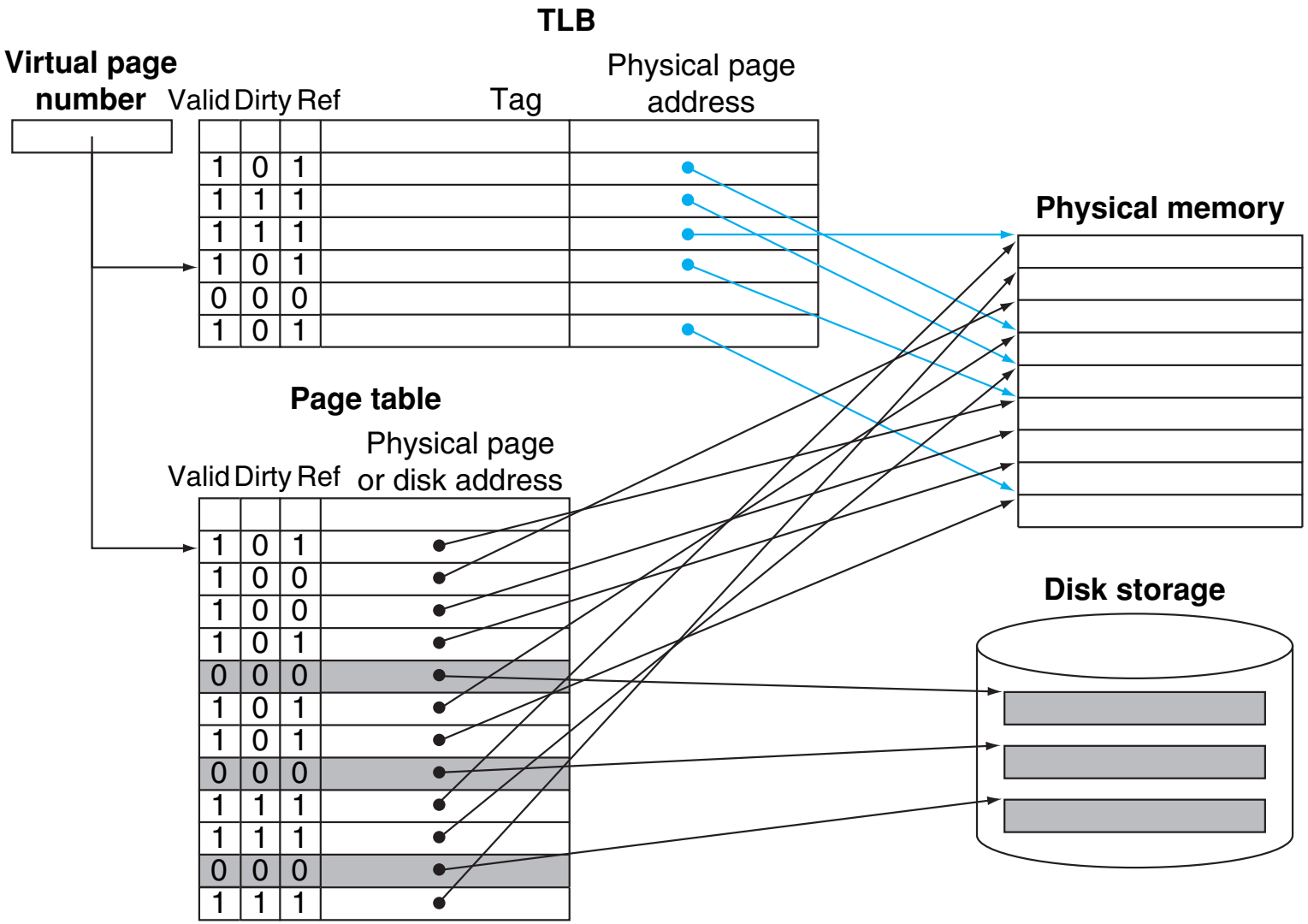

Virtual page number
Page table
Physical page or disk address
Valid
1
1
1
1
0
1
1
0
1
1
0
1
Physical memory
Disk storage

Some notes about performance

- Swapping pages between disk and physical memory is often referred to as *thrashing* and is extremely slow!
- The user observes this slow down when their computer does not have enough RAM and the hard drive activity light is continually flashing as they do work on the computer

More on performance

- The OS uses some form of Least Recently Used (LRU) to determine what goes in physical memory vs. disk
- In the previous diagram, every translation requires an additional memory access, which would be prohibitively slow
- Therefore, we would like a way to *cache* LRU portions of page table


TLB
Virtual page number
Valid Dirty Ref
Tag
Physical page address
1 0 1
1 1 1
1 1 1
1 0 1
0 0 0
1 0 1
Physical memory
Page table
Physical page or disk address
Valid Dirty Ref
1 0 1
1 0 0
1 0 0
1 0 1
0 0 0
1 0 1
1 0 1
0 0 0
1 1 1
1 1 1
0 0 0
1 1 1
Disk storage

The *Translation Lookaside Buffer (TLB)*

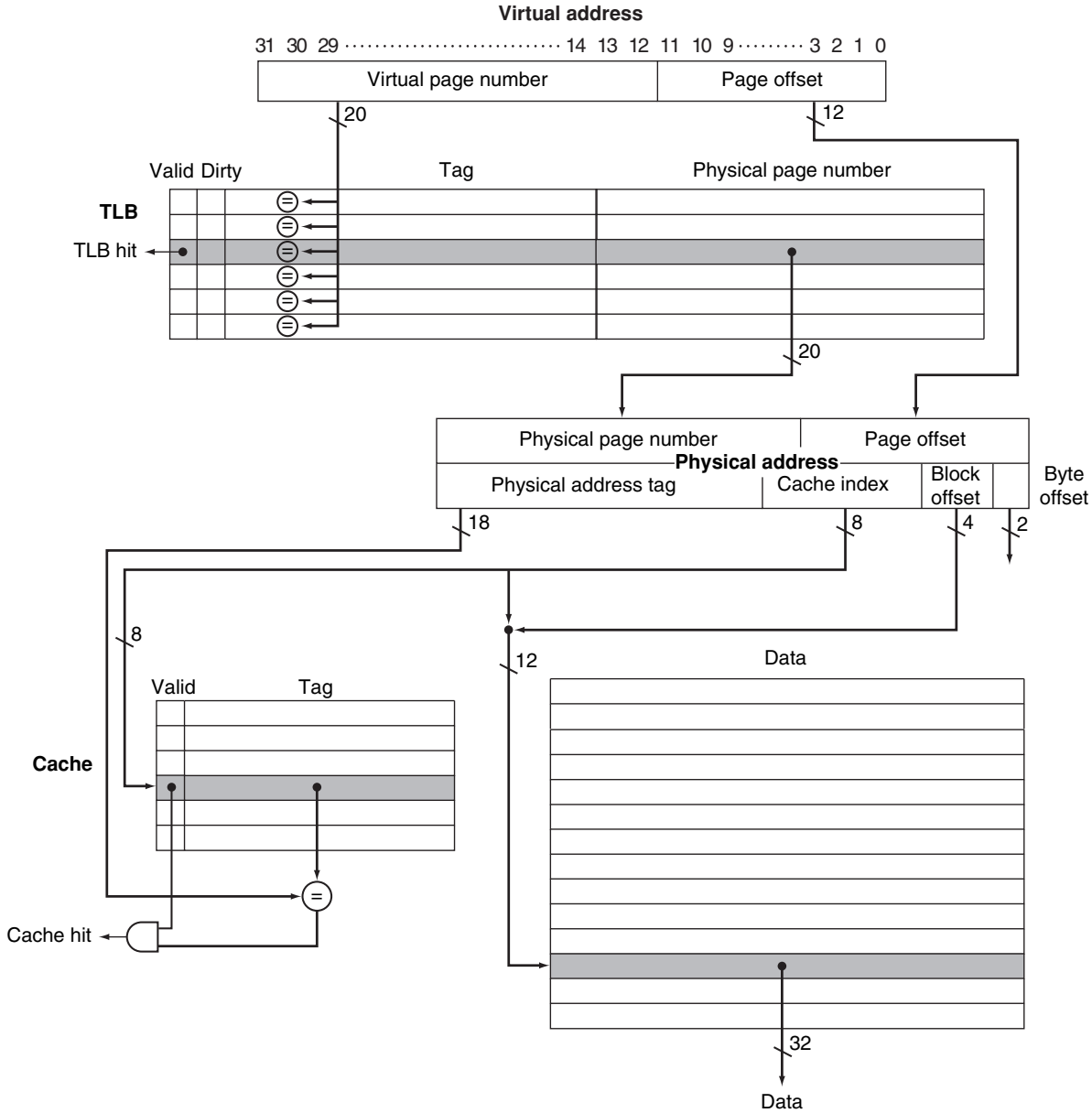
- The TLB is a small cache (typically with high maybe even full associativity) that stores most recently used page table entries
- The TLB, unlike the cache systems we have looked at so far, is managed by the OS
- A *TLB miss* occurs when a virtual page number is not found in the TLB
- The OS will then reference the page table to refill the TLB

More on the TLB

- Typical characteristics of a TLB:
  - Size: 16-512 entries
  - Block size: 1-2 page table entries (4-8 bytes each)
  - Hit time: 0.5-1 cc
  - Miss penalty: 10-100 clock cycles
  - Miss rate: 0.01%-1%
  - Block replacement: random
  - Associativity: fully
- Tightly integrated into the L1 caching system

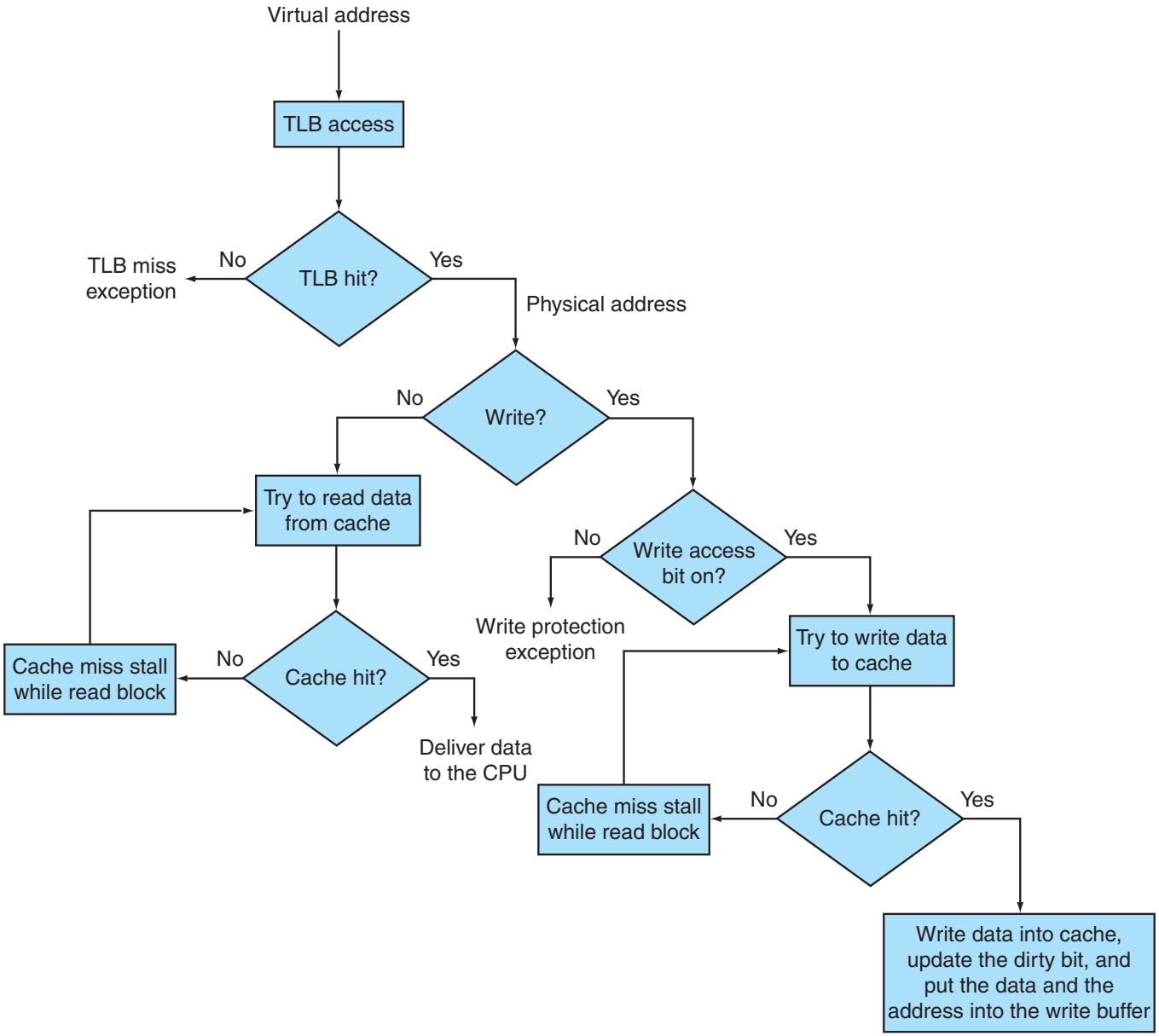
Let's look at the Intrinsity FastMath TLB

- Supports 4kB pages and a 32-bit address space
- 16 entry, fully associative, *shared* TLB
- 20-bit Virtual Page Number (VPN) and 20-bit Physical Page Number(PPN)
- 64-bit entries for VPN, PPN, valid bit, dirty bit, and access bits


Virtual address
31 30 29 ····· 14 13 12 11 10 9 ····· 3 2 1 0
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Block offset
Byte offset
18
8
4
2
8
12
Data
Valid
Tag
Cache
Cache hit
32
Data

A TLB Miss

- Occurs when the VPN is not found in the TLB
- A special TLB exception handler is invoked
  - Part of the OS
  - Accesses the page table to refill the TLB
  - Uses *write-back* policy to update dirty bit and access bits
- Takes 13 cc in the FastMATH assuming no cache misses


Virtual address
TLB access
TLB hit?
No
TLB miss exception
Yes
Physical address
Write?
No
Yes
Try to read data from cache
Cache hit?
No
Cache miss stall while read block
Yes
Deliver data to the CPU
Write access bit on?
No
Write protection exception
Yes
Try to write data to cache
Cache hit?
No
Cache miss stall while read block
Yes
Write data into cache, update the dirty bit, and put the data and the address into the write buffer

Possible combination of events

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

- Can't have a hit in the TLB but a miss in the page table!
- Can't have a hit in the cache but a miss in the page table

Additional support for virtual memory

- *User* and *kernel* modes and address regions
- Instructions and registers for maintaining the TLB (only accessible in kernel mode!)
- A *system call* exception, which transfers control from user mode to kernel mode
- An *unmapped* address region within kernel space for exception entries
- A way to enable and disable exceptions

Some MIPS control registers

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

The TLB miss handler

- Located at a special exception address dedicated for this purpose (fast!)
- Makes use of previously discusses registers and the `tlbwr` instruction
  - `tlbwr` loads the `EntryLo` and `EntryHi` contents into the TLB at a location pointed to by `Random`
  - `EntryLo` is the physical page address and flags from the Page Table Entry (PTE)
  - `EntryHi` is the virtual page address and is populated by the hardware

An example TLB handler

```
TLBmiss:
    mfc0 $k1, Context #grab PTE address
    lw $k1, 0($k1) #read page table
    mtc0 $k1, EntryLo
    tlbwr #write into TLB
    eret #return from exception
```

- A couple of notes
  - This TLB handler does not check to see if the PTE is valid!
  - The `Context` register is populated by the hardware during a TLB miss with the Address Space ID (ASID) and the upper bits of the `BadVAddr` register

The Address Space ID (ASID)??

- The system is running multiple *processes* each with its own *virtual address space*
- The ASID allows the hardware and OS to keep track of which particular address space it is working with
- Also, the TLB must store entries from various address spaces because *context switches* happen regularly; thus, it uses the ASID in addition the the VPN

More notes on the TLB miss handler

- Makes use the *k0* and *k1* registers
- The simple example shown previously only supports a single-level page table
- Linux uses a 3-level page table so the TLB miss handler for Linux is a bit more complicated
  - Executes about 20 instructions with four lw instructions and a handful of shifts and masking operations
  - Makes use of the `Context` and `BadVAddr` registers

The *unmapped* address region

- The unmapped address region within kernel space is necessary to avoid an exception happening when a TLB miss or other exception happens
- This region maps to the lowest region of *physical memory* by simply ignoring the upper bits of the virtual address
- A TLB miss vectors to 0x8000_0000 while all other exceptions including a page fault vector to 0x8000_0180

A general exception handler

- Handles page faults along with other aforementioned exceptions
- When an exception triggers, the CPU is automatically switched into kernel mode and *interrupts* are disabled
- Recall that the exception *vectors* to an address in unmapped kernel space which cannot generate a TLB exception

A general exception handler

- The handler pushes the General Purpose Registers (GPRs) and critical CPU registers onto the exception stack in kernel space
- Enables interrupts to allow nested exceptions and jumps to the C exception routine (in mapped/cached kernel space) which handles the rest of the exception processing
- When the C routine is finished, interrupts are disabled again, which means nested exceptions are not allowed for this short but critical moment

A general exception handler

- The GPRs and critical CPU registers are restored by popping them off of the exception stack
- An `eret` instruction is executed to use the `EPC` to return back to the original location that caused the exception
- A number of status mode bits work in concert to determine what privilege level the processor is at

Save state

Save GPR	addi	$k1,$sp, -XCPSIZE	# save space on stack for state
	sw	$sp, XCT_SP($k1)	# save $sp on stack
	sw	$v0, XCT_V0($k1)	# save $v0 on stack
	...		# save $v1, $ai, $si, $ti,... on stack
	sw	$ra, XCT_RA($k1)	# save $ra on stack
Save hi, lo	mfhi	$v0	# copy Hi
	mflo	$v1	# copy Lo
	sw	$v0, XCT_HI($k1)	# save Hi value on stack
	sw	$v1, XCT_LO($k1)	# save Lo value on stack
Save exception registers	mfc0	$a0, $cr	# copy cause register
	sw	$a0, XCT_CR($k1)	# save $cr value on stack
	...		# save $v1,....
	mfc0	$a3, $sr	# copy status register
	sw	$a3, XCT_SR($k1)	# save $sr on stack
Set sp	move	$sp, $k1	# sp = sp - XCPSIZE

Enable nested exceptions

	andi	$v0, $a3, MASK1	# $v0 = $sr & MASK1, enable exceptions
	mtc0	$v0, $sr	# $sr = value that enables exceptions

Call C exception handler

Set $gp	move	$gp, GPINIT	# set $gp to point to heap area
Call C code	move	$a0, $sp	# arg1 = pointer to exception stack
	jal	xcpt_deliver	# call C code to handle exception

Restoring state

Restore most GPR, hi, lo	move	$at, $sp	# temporary value of $sp
	lw	$ra, XCT_RA($at)	# restore $ra from stack
	...		# restore $t0,...., $a1
	lw	$a0, XCT_A0($k1)	# restore $a0 from stack
Restore status register	lw	$v0, XCT_SR($at)	# load old $sr from stack
	li	$v1, MASK2	# mask to disable exceptions
	and	$v0, $v0, $v1	# $v0 = $sr & MASK2, disable exceptions
	mtc0	$v0, $sr	# set status register

Exception return

Restore $sp and rest of GPR used as temporary registers	lw	$sp, XCT_SP($at)	# restore $sp from stack
	lw	$v0, XCT_V0($at)	# restore $v0 from stack
	lw	$v1, XCT_V1($at)	# restore $v1 from stack
	lw	$k1, XCT_EPC($at)	# copy old $epc from stack
	lw	$at, XCT_AT($at)	# restore $at from stack
Restore ERC and return	mtc0	$k1, $epc	# restore $epc
	eret	$ra	# return to interrupted instruction

A brief look at the *R4000*

- *The Mips R4000 Processor*, IEEE Micro 1992 by Sunil Mirapuri et al.
- Early 90s 64-bit MIPS processor
- Uses a technique called *super pipelining* with 8 stages instead of the traditional 5
- Supports 36-bit physical addresses, i.e. 64GB and a 64-bit virtual address space

A brief look at the *R4000*

- Virtually indexed, physically tagged L1 cache
- Has on-chip, split L1 caches with off-chip L2 cache support

The 8-stage pipeline

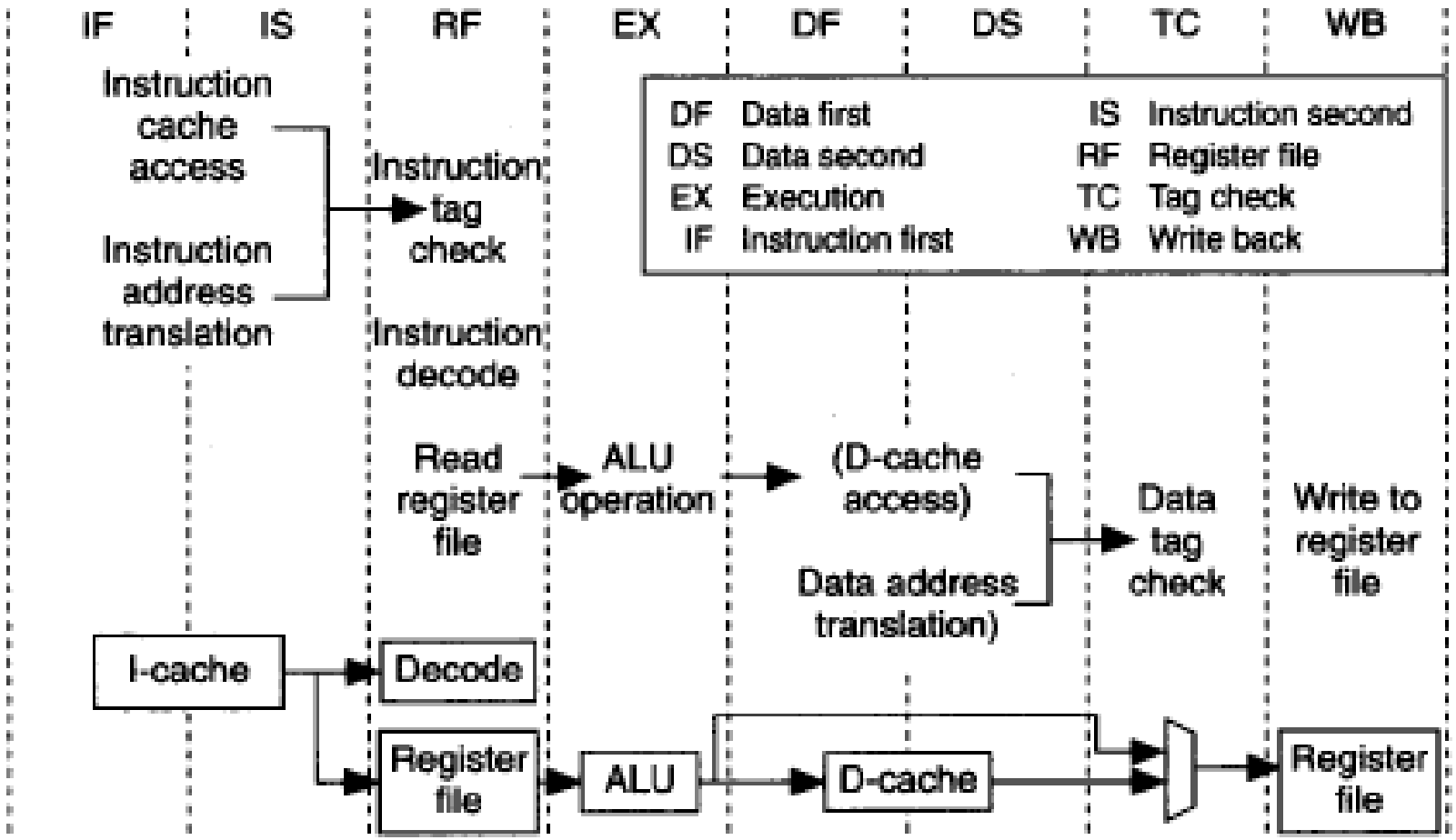
- Instruction cache access is in the Instruction First (IF) and Instruction Second (IS) stages
- Virtual to physical address translation happens also in these stages
- Decoding and register reading happen in the Register File (RF) stage
- Also, tag comparison for instruction fetch is performed in the RF stage

The 8-stage pipeline

- ALU operations take place in the Execution (EX) stage
- The authors state in the Micro paper that the EX stage is the critical path
- The TLB and data caches are accessed in the Data First (DF) and Data Second (DS) stages
- The Tag Check (TC) stage compares the tags

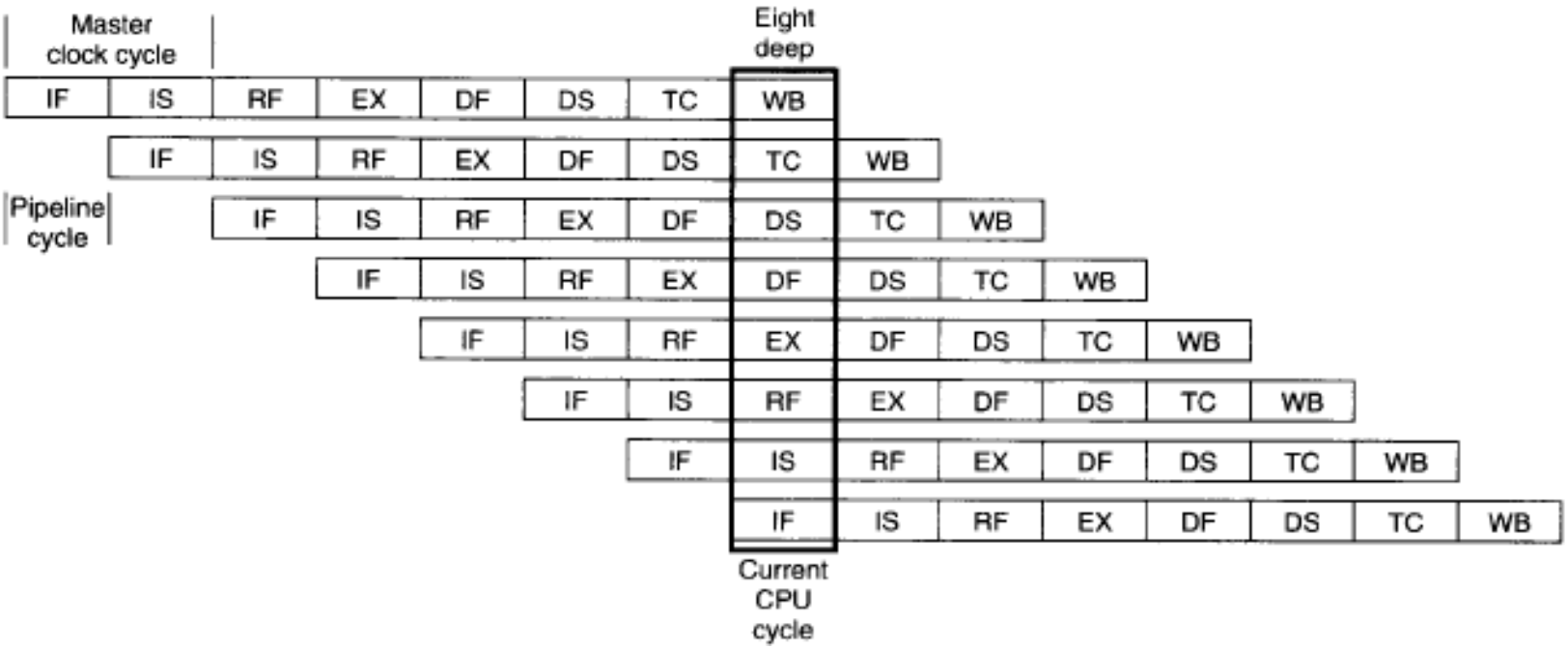
The 8-stage pipeline

- For stores, if the tags compare in the TC stage, the data is sent to the store buffer
- Data are written into the register file in the Write Back (WB) stage


IF
IS
RF
EX
DF
DS
TC
WB
Instruction cache access
Instruction address translation
Instruction tag check
Instruction decode
Read register file
ALU operation
(D-cache access)
Data address translation)
Data tag check
Write to register file
I-cache
Decode
Register file
ALU
D-cache
Register file
DF Data first
DS Data second
EX Execution
IF Instruction first
IS Instruction second
RF Register file
TC Tag check
WB Write back

So compared to the 5-stage pipeline

- The RF stage is like the ID stage in our 5-stage pipeline
- EX and WB are pretty much the same
- The IF stage was stretched out into IF and IS
- The MEM stage was stretched out into DF, DS, and TC
- The additional stages were for cache and TLB access


Master clock cycle
Eight deep
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
Pipeline cycle
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
IF IS RF EX DF DS TC WB
Current CPU cycle

Effects on CPI

- Branches are still resolved in the EX stage but now there is an addition instruction to flush
- Loads can now cause 2 stall cycles
- Note that the caches are direct mapped and data is forwarded before the TC stage. If the tags do not compare in the TC stage, the EX stage is repeated

L1 Caches

- Split 32kB, direct-mapped instruction and data caches
- Cache line size is programmable to either 16 or 32 bytes for both the instruction and data independently
- 2-entry store buffer avoids stalls for stores

L2 Cache

- Support for up to 4MB off-chip
- 128-bit wide interface can fill four word L1 cache line in one access
- Programmable line size of 4, 8, 16, or 32 *words*
- Error Correct Code (ECC) corrects 1-bit errors and detects 2-bit errors
- Multiprocessor support at the L2 level

Virtual memory support

- 48-entry TLB in which each entry mapping two consecutive pages (effectively 96-entries)
- The ITLB is a two-entry, fully associative buffer that is a subset of the main TLB
- ITLB only supports 4-kB pages
- TLB supports from 4kB to 16MB pages
- Data TLB access is pipelined across the DF/DS stages and runs in parallel with cache access

Operation Latencies

- Adds, subtracts, logic operations 1 cc
- Integer Multiply: 10 cc for 32-bit and 20 cc for 64-bit
- Integer divides: 69 cc for 32-bit, 133 cc for 64-bit
- Shifts: up to 32-bit 1 cc, greater than 32-bit 2 cc

How does it perform?

- The authors simulated the processor executing SPEC benchmarks assuming a 100MHz processor clock
- They looked at the performance using a 512KB L2, a 4MB L2, and no L2

Benchmark	S-cache size 4 Mbytes	S-cache size 512 Kbytes	P-cache only
Gcc	46	43	27
Espresso	54	54	38
Spice2g6	42	38	27
Doduc	49	46	33
Nasa7	56	46	43
Li	66	65	47
Eqntott	54	52	50
Matrix300	278	273	177
Fpppp	55	54	29
Tomcatv	58	59	37
Simulated SPEC	63	59	42
Simulated SPEC int	55	53	39
Simulated SPEC fp	69	64	44
CPI (simulated SPEC)	1.5	1.6	2.3

Shared Memory

- On-chip multi-core refers to multiple cores sharing memory on a single chip
- Multi-processor refers to multiple processors on separate chips sharing memory
- Multiple threads within a single process must have the same view of memory even when running on multiple cores and multiple processors

The challenge...

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

- In step 3, CPU A stores a 1 into X, which updates A's cache and memory (write-through), but not B's cache

Cache Coherency

- Ensuring processors sharing memory read the most recent value of a given memory location is referred to as *cache coherency*
- *write serialization* is another aspect of cache coherency in which every processor sees writes to a given location in the same order
- *consistency* is a more subtle aspect of shared memory that works with *coherency* to determine when exactly a processor will see a write into a shared memory location

What do shared caches provide?

- Cache coherency is maintained by the cooperation between *shared* and *private* caches
- Data *migration*, allowing threads to communicate via shared memory and transfer data back and forth
- Data *replication*, allowing threads to efficiently share data by storing their own copy of the data in their *private* caches

Cache coherency protocols

- Two main categories
  - *Directory* based in which a directory is maintained usually by the *shared* cache keeping track of the state of each cache block
  - *Snooping* base where each *private* cache snoops the bus to maintain the state of the each cache block
- Many systems will use a combination of directory and snooping

Snooping cache coherency protocol

- Snooping requires the use of a share bus with a broadcast medium
- When a cache block is retrieved by a private cache from a shared cache on the bus, all other private caches must see that read
- The same is true for a write

A simple *write invalidate protocol*

- A simple but fairly inefficient protocol is for each processor to send an invalidation message out on the bus every time it wants to write a block in the cache
- Every other processor sees this invalidation and invalidates that block if it's in its cache
- This maintains exclusive access to a block when writing

A simple *write invalidate protocol*

- When a processor modifies a block, all other processors will invalidate their copies
- When other processors read that block again, they will encounter a cache miss (a coherency miss to be specific!)
- The processor that wrote to the block will see the miss on the bus (snooping) and will provide that data on the bus, while forcing a write back into memory

A simple *write invalidate protocol*

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

- The invalidates enforce coherency, including the property of *write serialization*

More about the *write invalidate protocol*

- Although simple, it is inefficient because every write requires an invalidation message to be sent out on the bus to all other processors, regardless of whether or not anyone else is sharing the block
- We can improve the protocol by adding more states
  - Modified, Shared, Invalid (MSI)
  - Modified, Exclusive, Shared, Invalid (MESI)

Directory vs. Snooping

- Snooping is simple but requires a bus style interconnect with broadcast
- This limits the *scalability* of multi-core system
- A directory based protocol does not require a broadcast medium and can use other types of interconnection networks such as *mesh*
- A directory maintains the state of each cache and sends out invalidation messages only to those nodes that share the block

More about the *directory protocol*

- The directory is usually maintained by the share cache
- It can be distributed in the case of a distributed shared cache!
- It constrains the coherency messages to just those that share the given block

A quick note about cache block size

- Large blocks can lead to a greater number of *coherency misses*
- These misses are due to *false sharing*, in which data that is not actually being shared between processors but simply contained in the same cache block, causing unnecessary cache misses

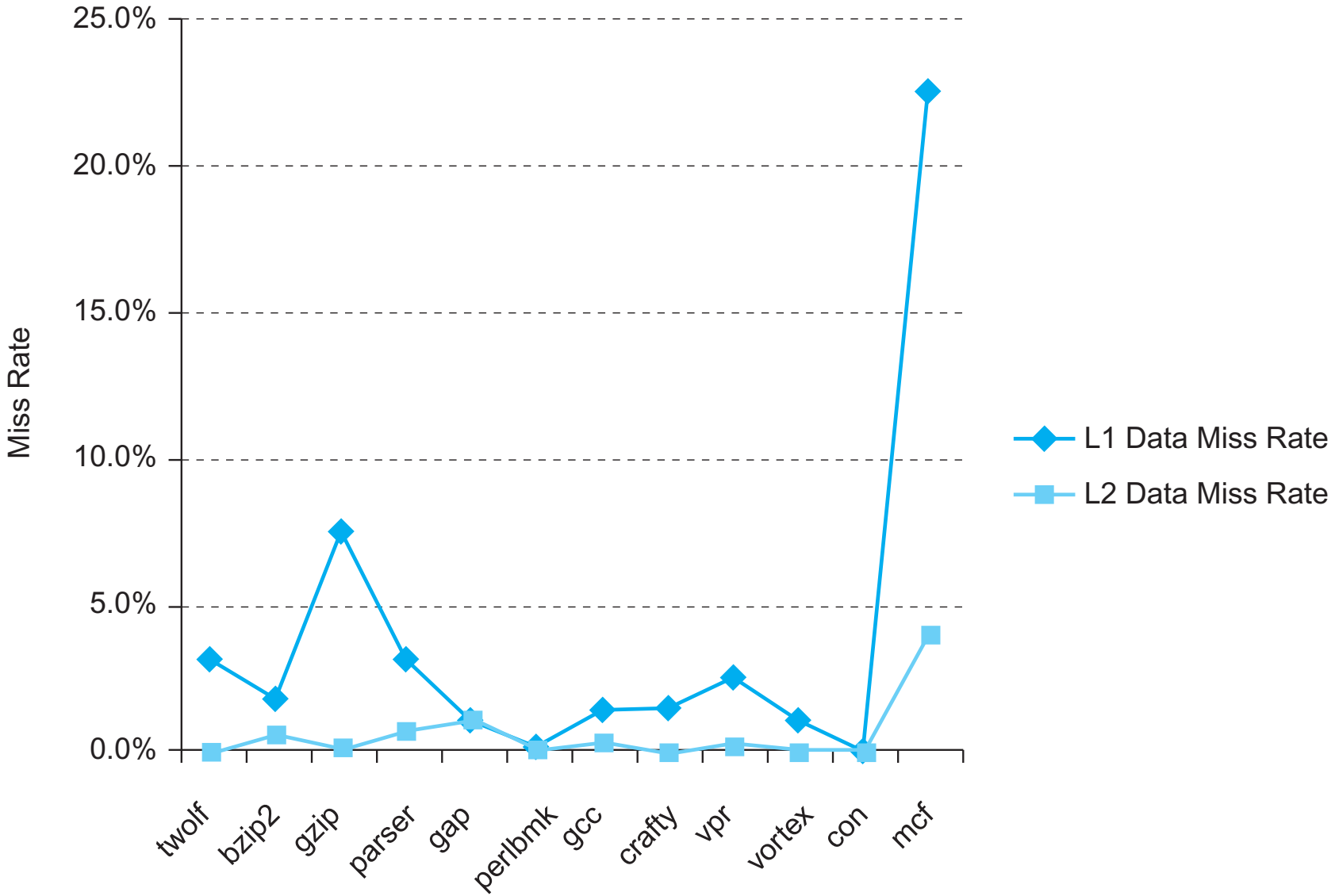
A comparison of the Cortex-A8 and Core i7

- The ARM Cortex-A8 is a 14-stage inorder pipelined processor used in many mobile devices
- The Intel Core i7 is a desktop grade 14-stage out-of-order pipelined processor

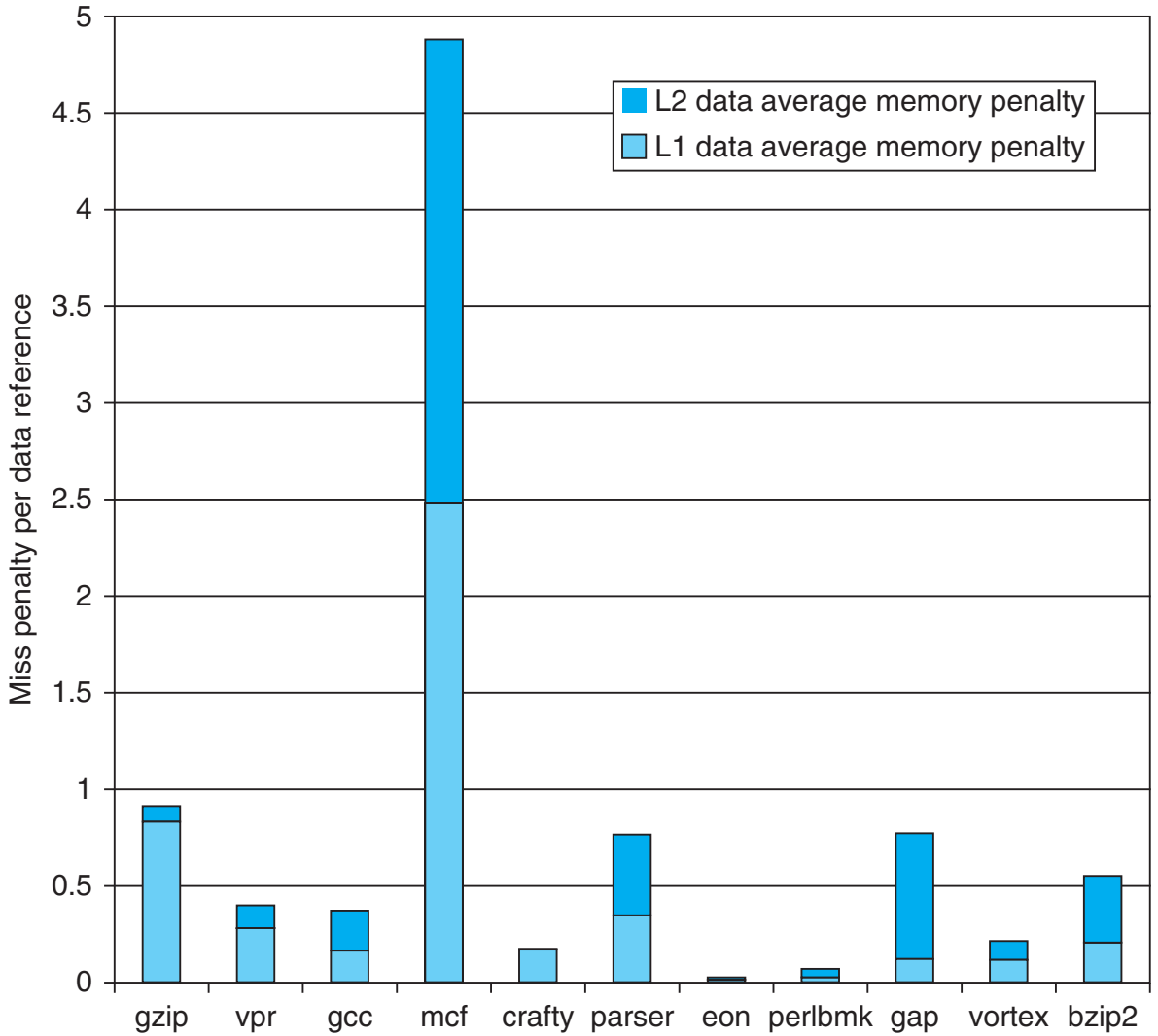
Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Cortex-A8 Miss Rate



Cortex-A8 average miss penalty



Core i7 Miss Rate